CMSC131

Mutability

How Objects Can (or Can't) Change

• When we apply methods such as `toUpperCase()` or `substring()` or even `concat(String)` to a `String`, we get a new `String` object returned. The original is unchanged.

• We have seen that even some commands that seem as though they would alter an object actually create a new one, like if we apply the `++` operator to an `Integer` object.

• However, we have written our own classes where the contents of an object can be changed via methods we create.
Mutability

• In object oriented languages, some object types are immutable (once the object is created, the values it holds can not be altered) while others are mutable.

• Different languages approach mutability in different ways, so it is important to explore the conventions of each language you learn.
  – In Objective-C there are usually both mutable and immutable versions of types.
  – Though not an object, it’s interesting to note that in Fortran, even literals like 1 weren’t always immutable!

Deep or Shallow Copies

• One place where we will need to think about whether something is mutable is when we talk about aliasing and making copies of things.

• We have seen this a little with our copy constructor and memory traces.

• Our Student class contained:
  ```java
  private String name;
  private String ident;
  private int tokenLevel;
  ```

• What does the following copy constructor do in terms of memory?
  ```java
  public Student(Student anotherStudent) {
    name = anotherStudent.name;
    ident = anotherStudent.ident;
    tokenLevel = anotherStudent.tokenLevel ;
  }
  ```
What if we *want* mutable strings?

• The `String` class in Java is immutable.
• The `StringBuffer` class in Java is mutable.

• Let's look at an example: `StringHolder`

---

The `StringBuffer` Class

• Some key methods we can use are:
  – `append`, which is overloaded in many ways
  – `insert`, which is also overloaded in many ways
  – `delete`, which allows you to delete any sub-part of the string
  – `replace`, which allows you to replace any sub-part of the string with another string

• These methods also return a *reference* to the `StringBuffer` being modified.

http://download.oracle.com/javase/6/docs/api/java/lang/StringBuffer.html
Mutability: Good or Bad?

• While a significant issue, some might argue it's not "too bad" since aliasing of mutable objects can be "solved" by making deep copies when needed.

• Are there any reasons why mutable objects would actually be good in their own right?

• Let's look at an example:
  
  `StringEfficiencyExample`

What are the trade-offs?

• What are the trade-offs involved in the previous example?
  
  – Time?
  
  – Space?
  
  – Other?
Copyright © 2010-2011 : Evan Golub